On Inspection and Verification of Software with Timing Requirements

Jia Xu

Abstract—Software with hard timing requirements should be designed using a systematic approach to make its timing properties easier to inspect and verify; otherwise, it may be practically impossible to determine whether the software satisfies the timing requirements. Preruntime scheduling provides such an approach by placing restrictions on software structures to reduce complexity. A major benefit of using a preruntime scheduling approach is that it makes it easier to systematically inspect and verify the timing properties of the actual software code, not just various high-level abstractions of the code.

Index Terms—Real-time, software, code, inspection, verification, timing requirements, current practices, complexity, restrictions, software structures, preruntime scheduling, predictability.

1 Introduction

More and more infrastructure of the world is becoming dependent on computer systems that have timing requirements. Communications, transportation, medicine, energy, finance, and defense are all increasingly involving processes and activities that require precise observance of timing constraints. Real-time software is often required to handle the coordination and synchronization of many different processes and activities. As a result, real-time and embedded software that must observe timing constraints is experiencing explosive growth.

In contrast, there is a conspicious lack of effective methods and tools for verifying timing properties of software, despite an increasingly pressing need for such methods and tools.

Examples of proposed formal methods for real-time systems, include, among others, timed and hybrid automata [2], [1]; timed transition systems/temporal logic [33], [23], [41], [29], [7]; timed Petri-nets [26]; theorem proving techniques using PVS to analyze real-time protocols and algorithms [21]. The most industrialized of the formal methods is model checking [22], [44], [11]. While formal methods have been used successfully for verifying hardware designs, the use of formal methods to verify actual software code is rare [13], [20], [27], [17], [42], [31] and the use of formal methods for verifying timing properties of actual large scale real-time software implementations (as opposed to simplified high-level abstractions of code such as specifications/models/algorithms/protocols which are only approximations of the actual software and which do not take into account all the implementation details that may affect timing) is practically nonexistent.

- *The author is with the Department of Computer Science, York University, 4700 Keele St., North York, Ontario M3J 1P3, Canada. E-mail: jxu@cs.yorku.ca.*

Manuscript received 8 Sept. 2002; revised 20 Feb. 2003; accepted 27 Mar. 2003.
Recommended for acceptance by D. Parnas and M. Lawford.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 118511.

What is the main reason for this apparent difficulty in developing effective methods and tools for verifying timing properties of software? The problem is the complexity of software, especially nonterminating concurrent software, and the complexity of such software's possible timing behaviors.

Timing requirements and constraints in concurrent, real-time software pose special problems for software inspection and verification. The basic premise of any software inspection or verification method is that it should be able to cover all the possible cases of the software's behavior. Taking into account timing parameters and constraints adds a whole new dimension to the solution space. The number of different possible interleaving and/or concurrent execution sequences of multiple threads of software processes and activities that need to be considered when timing constraints are included may increase exponentially as a function of the size of the software and may result in an explosion of the number of different cases that needs to be examined. This may make it exceedingly difficult to use verification and inspection techniques that systematically examine all the possible cases of software behavior.

The next section provides further discussion on why timing properties of software are difficult to inspect and verify. Section 3 examines current practices in the design of real-time software that make it difficult to inspect and verify timing properties. Section 4 describes a preruntime scheduling approach for structuring real-time software that will help simplify inspection and verification of timing through a detailed example. Section 5 discusses how the preruntime scheduling approach may help to alleviate the theoretical limitations and practical difficulties in verifying timing properties in software. Conclusions are presented in Section 6.

2 Why Is Verifying Timing Properties of Software Difficult?

In the following, we discuss why verifying timing properties of software in general is difficult.

0098-5589/03/$17.00 © 2003 IEEE

2.1 Fundamental Theoretical Limitations

Fundamental theoretical limitations in verifying timing properties of software include the following:

1. Even without considering timing properties, the problem of program verification (i.e., is a given program correct with respect to a given specification) is in general *undecidable*. Many of the proposed logics and models of real-time are undecidable [7], [30]. The ability to express timing properties in general increases the complexity of a logical theory. For example, first-order (discrete time) temporal logic is undecidable in the general case mainly due to the quantification of time-dependent variables [3]. Even restricted computable versions of the problem (e.g., finite state, discrete time domain, and prohibiting quantification on time variables) are of high complexity, being at least PSPACE-complete and, more often, EXPSPACE-complete [4]. Methods and tools based on these logics and models are in general subject to the *state space explosion* problem, i.e., the state space size grows exponentially with the size of the program description.

   For example, the most successful of the formal methods, model checking, exhaustively enumerates the state space. Because the size of the state space in general grows exponentially with the size of the description of the model, despite the use of symbolic model checking methods [37][1] and other special techniques, model checking has mainly been limited to checking properties of hardware circuits. Attempts to use symbolic model checking to check timing properties have been restricted to high-level algorithms/protocols [9], which are much less complex than software. Most of the success of model checking is not so much in the formal verification of specifications, but in the finding of bugs not found by other informal methods, such as testing and simulation, through exploring only part of the state space [17].
2. When verifying software with timing requirements, it is difficult to separate timing correctness and logical correctness. Timing correctness depends on logical correctness because logical errors can cause timing errors. For example, deadlocks, stack/buffer overflows, starvation of processes, divide-by-zero exceptions, priority inversion, infinite loops, erroneous memory references, etc., may cause unexpected time delays. Logical correctness can also depend on timing correctness because timing errors can cause logical errors. For example, a process that unexpectedly fails to complete some time-critical function due to a timing error, may leave the system in an unexpected erroneous state. Because program logical correctness is undecidable and timing correctness depends on logical correctness, this is one of the reasons that program timing correctness in general is also undecidable.
3. Because program verification is undecidable, program verification methods are all partial or incomplete. In order to cope with undecidability or complexity, all program verification methods use some form of *approximation*. The approximation may take many different forms. In model checking and other verification methods, often the model only preserves selected characteristics of the implementation while abstracting away complex details. When *abstraction* is used to approximate the program behavior, it becomes necessary to prove that the abstraction is sound, i.e., the questions about the program can be answered correctly with the abstraction despite the loss of information. But, the discovery and proof of soundness of the required abstraction is logically equivalent to a direct correctness proof of the program itself [16]. So far, techniques for discovering and proving the soundness of required abstractions for the verification of timing properties of software in general are not available. Most of the work related to verification of timing properties only studies simplified high-level abstractions of algorithms/protocols which are only approximations of the actual software and which do not take into account all the implementation details that may affect timing.

1. With symbolic model checking, transition relations and sets of states are represented symbolically using Boolean functions called Ordered Binary Decision Diagrams (BDDs). Hardware circuits are often regular and hierarchical, thus, in many practical situations, the space requirements for the Boolean functions representing the hardware circuits are much smaller than for explicit represention, alleviating the space explosion problem in those situations. However, as explained in [12], practical experiments show that the performance of symbolic methods is highly unpredictable, which can be partially explained by complexity theoretical results which state that BDD representation does not improve worst-case complexity; in fact, representing a decision problem in terms of exponentially smaller BDDs usually increases its worst-case complexity exponentially.

2.2 Practical Difficulties

Practical difficulties in verifying timing properties of software include the following:

1. External events in the environment and internal events in a computer system may happen nondeterministically at any time. Thus, asynchronous processes that respond to those events may request execution nondeterministically at any time, and the control structure of the software may allow those processes to execute nondeterministically at any point in time and interrupt/preempt other processes nondeterministically at any point in time and at any point in the logical control flow. When the software exhibits nondeterministic behavior, the complexity of the logical and timing behaviors of the system increases significantly.

   For example, the developers of TAXYS, a state-of-the-art tool which uses the formal model of timed automata [2] and uses the KRONOS model checker [18] for verifying timing properties of real-time embedded systems, reported experimental results in which the tool was forced to abort when the number of symbolic states explored by KRONOS increased exponentially with the "degree" of nondeterminism, even though the system being verified

contained only two strictly periodic, independent tasks and one aperiodic (asynchronous) task [14].

2. Unless information about the runtime resource usage pattern can be known before runtime with sufficient accuracy and is taken into account, assertions about the timing properties that can be verified will often be overly pessimistic.

   For example, currently, many modern processors use technologies such as pipelines and cache hierarchies to increase average-case performance, but such technologies that increase the average-case performance may increase the worst-case execution times. Without more accurate information about the runtime execution sequences of machine instructions known before runtime, it would be difficult to more accurately predict the worst-case execution times of the runtime execution sequences and avoid overly pessimistic predictions when such technologies are used.

3. The more detailed the hardware/software characteristics that are taken into account, the higher the precision with which timing properties can be verified. When it is necessary to verify timing properties with high precision, e.g., when time bounds are tight, it is necessary to verify timing properties at a low level. It may even be necessary to verify timing properties at the implementation machine code/assembler level in order to achieve sufficient precision. If the software has a high complexity, then this may increase the size of the state space that needs to be examined to the extent that it is impossible to do so within practical time/memory limitations.

3 Current Practices in the Design of Real-Time Software that Make It Difficult to Inspect and Verify Timing Properties

In the following, we mention current practices in the design of real-time nonterminating and concurrent software that make it more difficult to verify and inspect timing properties, very much like the unrestricted use of "goto" statements destroy the structures of regular programs and make it more difficult to inspect and verify their properties.

1. Complex synchronization mechanisms are used in order to prevent simultaneous access to shared resources. These synchronization mechanisms often use queuing constructs where queuing policies such as FIFO and blocking can make the timing behavior unpredictable.
2. Real-time processes not only execute at random times, they are often allowed to preempt other processes at random points in time. Not only the context switch times vary, but it also results in a huge increase in the number of different possible execution interleaving sequences, many of which may have unpredictable effects on timing.
3. The execution of runtime schedulers and other operating system processes such as interrupt handling routines with complex behaviors (and often with the highest priorities [34]) interleave with the execution of real-time application processes, affecting the timing of the application processes in subtle and unpredictable ways.
4. When many additional constraints are required by the application, such as precedence constraints, release times that are not equal to the beginning of their periods, low jitter requirements,[2] etc., current runtime scheduling algorithms and mechanisms are unable to solve problems that include these additional constraints, practitioners use ad hoc runtime methods to try to satisfy the additional constraints. These ad hoc runtime methods tend to affect timing in highly unpredictable ways.
5. Fixed priorities are used to try to deal with every kind of requirement [34]. The fixed priority assignments often conflict with other application requirements. In practice, task priorities are rarely application requirements, but are used instead as the primary means for trying to meet timing constraints. These priorities frequently change, which greatly complicates the timing analysis.
6. Task blocking is used to handle concurrent resource contention, which, in addition to making the timing unpredictable, may result in deadlocks.

Even in fairly simple systems in which a few of the above practices are used, inspecting software with timing constraints can still be a very daunting task. For example, in one study, fixed priority scheduling was implemented using priority queues, where tasks were moved between queues by a scheduler that was run at regular intervals by a timer interrupt. It had been observed that, because the clock interrupt handler had a priority greater than any application task, even a high-priority task could suffer long delays while lower priority tasks were moved from one queue to another. Accurately predicting the scheduler overhead proved to be an extraordinarily complicated task, even though the system was very simple, with a total of only 20 tasks, where tasks do not have critical sections, priorities do not change, and the authors of the study are considered to be among the world's foremost authorities on fixed priority scheduling [8].

When the above current practices are used in combination with each other, the high complexity of the interactions between the different entities and the sheer number of different possible combinations of those interactions significantly increase the chances that some important cases will be overlooked in the inspection and verification process.

The fundamental theoretical limitations discussed in the previous section tell us that, for a given collection of software and a given set of timing requirements, if the software and its timing behaviors are overly complex, then it may be practically impossible to determine whether the software satisfies the timing requirements.

Imposing restrictions on software structures to reduce complexity seems to be the key to constructing software so that timing properties can be easily inspected and verified.

2. *Jitter* refers to the variation in time a computed result is output to the external environment from cycle to cycle.

Software with hard timing requirements should be designed using a systematic approach to make their timing properties verifiable and easy to inspect. There will probably never exist a general method or tool that can verify the timing properties of any arbitrary piece of software, just like it is unlikely that general methods or tools will prove effective in verifying properties of a badly structured program consisting of "spaghetti" code woven together with "goto" statements.

4 A Preruntime Scheduling Approach for Structuring Real-Time Software that Will Simplify Inspection and Verification of Timing

Below, we describe a *preruntime scheduling approach* for structuring real-time software which will simplify inspection and verification of timing because the approach imposes strong restrictions on the structure of the software to reduce its complexity.

Without loss of generality, suppose that the software we wish to inspect for timing consists of a set of sequential programs. Some of the programs are to be executed periodically, once in each period of time. Some of the programs are to be executed in response to asynchronous events. Assume also that, for each periodic program p, we are given the earliest time that it can start its computation, called its *release time* r_p, the *deadline* d_p by which it must finish its computation, its *worst-case computation time* c_p, and its *period* prd_p. For each asynchronous program, we are given its *worst-case computation time* c_a, its *deadline* d_a, and the *minimum time between two consecutive requests* min_a. Furthermore, suppose there may exist some sections of some programs that are required to *precede* a given set of sections in other programs. There also may exist some sections of some programs that *exclude* a given set of sections of other programs. Also, suppose that we know the computation time and start time of each program section relative to the beginning of the program containing that section. It is assumed that the worst-case computation time and the logical correctness of each program has been independently verified.[3]

The preruntime scheduling approach consists of the following steps:

1. Organize the sequential programs as a set of cooperating sequential processes to be scheduled before runtime.
2. Identify all critical sections, i.e., sections of programs that access shared resources, as well as any sections of a program that must be executed before some sections of other programs, such as when a producer-consumer type of relation exists between sections of programs.

   Divide each process into process segments such that appropriate exclusion and precedence relations can be defined on pairs of sequences of the process segments to prevent simultaneous access to shared resources and ensure proper execution order.
3. Convert asynchronous processes into periodic processes.[4] For each asynchronous process a with computation time c_a, deadline d_a, and a minimum time between two consecutive requests min_a, convert it into a new periodic process p with release time r_p, computation time c_p, deadline d_p, and period prd_p that satisfies the following set of general conditions [39], [40].

   - $c_p = c_a$.
   - $d_a \geq \ d_p \geq c_a$.
   - $prd_p \leq \min(d_a - d_p + 1, min_a)$.

   Note that the above general conditions provide a range of possible values, instead of one unique value, for d_p and prd_p, respectively.

   The following set of conditions heuristically selects one of the possible values, for d_p and prd_p, respectively:[5] Suppose that the existing set of periods of periodic processes is P.

   a. $r_p = 0$.
   b. $c_p = c_a$.
   c. prd_p is equal to the largest member of P such that $2 \times prd_p - 1 \leq \ d_a$ and $prd_p \leq \ min_a$.
   d. d_p is equal to the largest integer such that $d_p + prd_p - 1 \leq d_a$ and $d_p \leq prd_p$.
   e. $d_p \geq \ c_a$.
4. Calculate the release time and deadline for each process segment.

   For each process p with release time r_p, deadline d_p, and consisting of a sequence of process segments $p_0, p_1, \ldots, p_i, \ldots, p_n$, with computation times $c_{p_0}, c_{p_1}, \ldots, c_{p_i}, \ldots, c_{p_n}$, respectively, the release time r_{p_i} and deadline d_{p_i} of each segment p_i can be calculated as follows:

$$r_{p_i} = r_p + \sum_{j=0}^{i-1} c_{p_j} \qquad d_{p_i} = d_p - \sum_{j=i+1}^{n} c_{p_j}$$

5. Compute a schedule offline, called a *preruntime schedule*, for all instances of the entire set of periodic segments, including new periodic segments converted from asynchronous segments, occurring

3. Here, we use the principle of "separation of concerns" [19]. By organizing the software as a set of cooperating sequential processes, thus eliminating concurrency within each process, we reduce the complexity of the software by making it both easier to independently verify the worst-case execution time and logical correctness of each process and easier to verify the correctness of the timing and logical correctness of all the possible interactions between the processes. This is further discussed briefly in Section 6.

4. If some asynchronous process cannot be converted into a corresponding periodic process using the heuristic given here, i.e., no corresponding periodic process exists that satisfies the set of conditions given here, then one may try using the procedure for converting asynchronous processes into periodic processes that is described in [53]. For asynchronous processes with very short deadlines, it may be necessary to let such asynchronous processes remain asynchronous and use an algorithm such as that in [53] that integrates runtime scheduling with preruntime scheduling to schedule the unconverted asynchronous processes at runtime.

5. The set of conditions given here includes elements of the heuristics described in [39] and [40]. The heuristic described in [39] is likely to produce new periodic processes for which the deadline exceeds the period, while the heuristic in [40] may result in a deadline that is too short and may conflict with other processes. The set of conditions in this heuristic were chosen to address such concerns. One should note that no single heuristic for converting a given set of asynchronous processes into periodic processes is optimal for the schedulability of an arbitrary set of processes.

within a time period that is equal to the least common multiple[6] of all periodic segments, which satisfies all the release time, deadline, precedence, and exclusion relations [51], [53], [54], [55], [56].

6. At runtime, execute all the periodic segments in accordance with the previously computed schedule.

   The following is just one of the many possible ways of executing process segments in accordance with the preruntime schedule.

   The system includes a timer, an interrupt mechanism, and a preruntime schedule dispatcher.

   a. Data structures include:

      - a Preruntime Schedule Table, each entry containing information about the start time of each segment/subsegment in the preruntime schedule, a pointer to a function containing the code of the segment/subsegment, a pointer to the next segment/subsegment in the schedule, etc.
      - a set of global variables containing information about where to find the Preruntime Schedule Table, length of the preruntime schedule, the time that the preruntime schedule was first started, pointers pointing to entries in the Preruntime Schedule Table to indicate which segment/subsegment is currently executing and which segment/subsegment is to be executed next, etc.

   b. If a process/segment is to be preempted according to the preruntime schedule, then, prior to runtime, in the case of a preempted segment, the segment is divided into separate subsegments according to the preruntime schedule, and information about each subsegment is entered into the Preruntime Schedule Table mentioned above; context saving code is appended to the end of the subsegment or the end of the segment in a preempted process that immediately precedes the preemption point; context restoring code is inserted at the beginning of the subsegment or the segment that should resume from the preemption point. At runtime, a resume time of a subsegment can then be treated the same way as a start time of a segment for dispatching purposes, allowing subsegments to be dispatched in a way that is essentially similar to segments.
   c. Code that transfers control to the dispatcher is appended to the end of each segment/subsegment prior to runtime.
   d. In general, the timer is always programmed to interrupt at the next starting time or resume time of a segment/subsegment according to the information in the Preruntime Schedule Table.
   e. At each timer interrupt, control is transferred to the dispatcher, which first checks whether the previously executing segment/subsegment has completed its computation or not; in case a segment/subsegment has not completed, it will perform error handling and recovery; otherwise, it will use the data structures in Step 6.a to determine which segment should be executed next. Before transferring control to that segment, the dispatcher reprograms the timer to interrupt at the next starting time or resume time of a segment according to the information in the Preruntime Schedule Table. The dispatcher will execute a segment that corresponds to a converted asynchronous process only if an asynchronous request has been made for the execution of that process.
   f. If a segment/subsegment completes its computation before the end of the time slot allocated to it in the preruntime schedule (which would be the normal case since the time slot lengths correspond to the worst-case computation times), control would be transferred back to the dispatcher through the code mentioned in Step 6.c. In this case, the dispatcher may use the unused time in the current time slot to execute other processes that are not time critical or idle the processor if there are no such processes.

   (The non-time-critical processes will transfer control back to the dispatcher on completion through code similar to that mentioned in Step 6.b, or will be interrupted by a timer interrupt before the next start time or resume time of a segment. General context saving and restoring when non-time-critical tasks are preempted can be performed by the timer interrupt handler or dispatcher. The timer should be programmed to interrupt noncritical tasks earlier than the start times of segments so that there is sufficient time to perform context saving for noncritical tasks and so that the running of non-time-critical tasks will not interfere with the timely execution of time-critical segments.)

   The worst-case overhead required by the interrupt handler, dispatcher, context saving/restoring code, and code for transferring control back to the dispatcher described above, etc., should be included in the worst-case computation times of the processes/segments prior to Step 3 in the preruntime scheduling approach.

   The above scheme will guarantee that every segment/subsegment will be executed strictly within the time slot that was allocated to it in the preruntime schedule.

It is noted here that there are many alternative ways of executing segments in accordance with a preruntime schedule. For example, instead of appending context saving/restoring code to segments/subsegments at preemption points, one can store the context saving/restoring

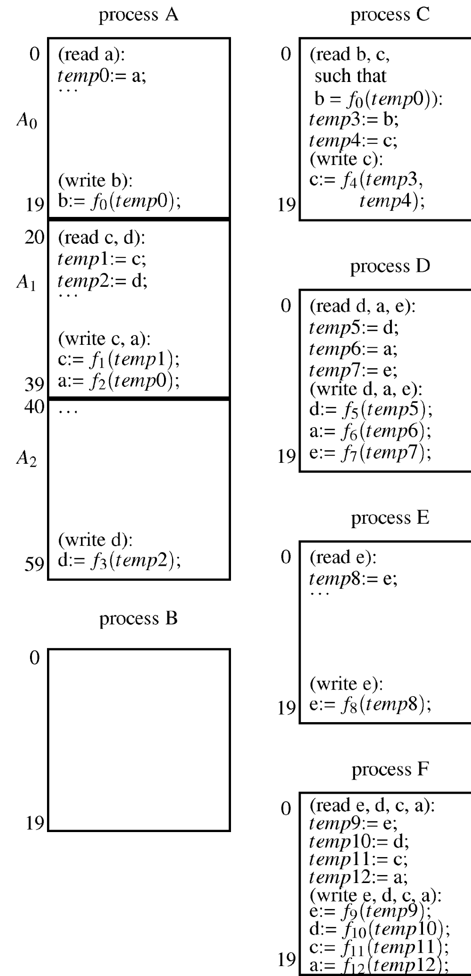
6. When the process periods are relatively prime, the Least Common Multiple (LCM) of the process periods and the length of the preruntime schedule may become inconveniently long. However, in practice, one often has the flexibility to adjust the period lengths in order to obtain a satisfactory length of the LCM of the process periods. While this may result in some reduction in the processor utilization, the reduction should be insignificant when compared to the decrease in processor utilization with fixed priority scheduling.

code elsewhere and include a pointer to the context saving/restoring code in the entry of the Preruntime Schedule Table corresponding to each of those segments/subsegments, then let the dispatcher use that information to perform the corresponding context saving/restoring if necessary. Another example is, if a subsegment/segment in the preruntime schedule does not have an exclusion relation or precedence relation with some other segment that may be violated if that subsegment/segment is executed earlier than the beginning of its time slot, then the preruntime schedule dispatcher could allow that subsegment/segment to be executed as soon as the preceding subsegment/segment belonging to the same process finishes and program the timer to interrupt it if it has not completed by the next start time of a segment according to the preruntime schedule. In such cases, the context saving could be handled by the timer interrupt handler, and the context restoring and resumption of the remaining part of the subsegment/segment could be handled by the dispatcher. It is also possible to execute segments/subsegments in the preruntime schedule through program structures and mechanisms similar to procedure calls. The details/advantages/disadvantages of each alternative way is beyond the scope of this paper and will be discussed in detail in a forthcoming separate paper.[7]

In the following, we will show how to perform preruntime scheduling through an example.

We note that neither existing scheduling or synchronization mechanisms that schedule all processes at runtime nor previous work on static scheduling would be able to guarantee that the set of processes in this example would always satisfy their timing constraints. In contrast, we will show that, with a preruntime scheduling approach, one can easily guarantee that the set of processes in this example would always satisfy their timing constraints.

Example 1. Suppose that, in a hard-real-time system, the software consists of six sequential programs A, B, C, D, E, and F, which have been organized as a set of cooperating sequential processes that cooperate through reading and writing data on a set of shared variables a, b, c, d, and e. Among them, A, B, C, and D are to be executed periodically, with release times at 0, 20, 30, and 90 time units; computation times 60, 20, 20, and 20 time units; deadlines 120, 120, 50, and 110 time units; periods 120, 240, 120, and 240 time units, respectively. Programs E and F are to be executed in response to asynchronous requests, with computation times at 20 and 20 time units, deadlines 480 and 481 time units, and minimum time between two consecutive requests 242 and 243 time units, respectively. The release time indicates the earliest time at which a process can start its computation. The deadline indicates the time by which each process must have completed its computation.



r[A] = 0	r[B] = 20	r[C] = 30	r[D] = 90		
c[A] = 60	c[B] = 20	c[C] = 20	c[D] = 20	c[E] = 20	c[F] = 20
d[A] = 120	d[B] = 120	d[C] = 50	d[D] = 110	d[E] = 480	d[F] = 481
prd[A] = 120	prd[B] = 240	prd[C] = 120	prd[D] = 240	min[E] = 242	min[F] = 243

Fig. 1. The sequential programs A, B, C, and D are to be executed periodically. The sequential programs E and F are to be executed in response to asynchronous requests.

Process A, at the beginning of its computation, reads the current value of the shared variable a and proceeds to perform a computation based on that value. At the 20th time unit of its computation, process A writes a new value into another shared variable b; that value depends on the previously read value of a. This new value of b is intended to be read by process C. At the 21st time unit of its computation, process A also reads the current values of two other shared variables c and d, and, at the 40th time unit of its computation, writes new values into c and a, where the new values depend on the previously read values of c and a, respectively. At the 60th time unit, that is, at the last time unit of its computation, process A writes a new value into d; the new value depends on the previously read value of d. (See Fig. 1.)

Process B performs some computation that does not read or write any variables that are shared with any other processes.

7. The actual implementation can be performed in a reasonable amount of time. Graduate students and fourth year students have completed course projects involving the implementation of such preruntime schedule dispatchers as loadable kernel modules, replacing the original scheduler and timer interrupt handler of the RTAI real-time operating system in a one semester course taught by this author [48]. In some of the student projects, time that is not used by time-critical segments in the preruntime schedule is used by the runtime dispatcher to run Red Hat Linux as a low priority preemptable process, which in turn runs its own scheduler to run Linux (non-time-critical) tasks.

Process C, at the beginning of its computation, reads the current values of the shared variables b and c. The value of b is required to be that produced by process A. At the end of its computation, process C writes a new value, which depends on the previously read value of b and c, into c.

Process D, at the beginning of its computation, reads the current values of the three shared variables d, a, and e, performs a computation, and, at the end of its computation, writes new values into d, a, and e that depend on the previously read values of d, a, and e, respectively.

Process E, at the beginning of its computation, reads the current values of the shared variable e, performs a computation, and, at the end of its computation, writes a new value into e that depends on the previously read value of e.

Process F, at the beginning of its computation, reads the current values of the four shared variables e, d, c, and a, performs a computation, and, at the end of its computation, writes new values into e, d, c, and a that depend on the previously read values of e, d, c, and a, respectively.

In order to prevent processes from simultaneously accessing shared resources, such as the shared data in this example, we can divide each process into a sequence of segments and define *critical sections*, where each critical section consists of some sequence of the process segments. We then define "EXCLUDES" (binary) relations between critical sections. We require of any satisfactory schedule that the following condition be satisfied for excludes relations on critical sections: *For any pair of critical sections x and y, if x $EXCLUDES$ y, then no computation of any segment in y can occur between the time that the first segment in x starts its computation and the time that the last segment in x completes its computation.* Note that x $EXCLUDES$ y does not imply y $EXCLUDES$ x, thus, if mutual exclusion between two critical sections x and y is required, then one should specify two separate exclusions: x $EXCLUDES$ y and y $EXCLUDES$ x.

In order to enforce the proper ordering of segments in a process, as well as producer/consumer relationships between segments belonging to different processes, we can define a *"precedes"* relation "PRECEDES" on ordered pairs of segments. We require that the following conditions be satisfied for precedes relations on segments: *For any pair of segments i and j, if i PRECEDES j, then segment j cannot start its computation before segment i has completed its computation.*[8]

In the example above, in order to prevent simultaneous access to the shared variables a, b, c, d, and e, enforce the required producer/consumer relationship between process A and C, while, at the same time, maximize the chances of finding a feasible schedule, one can do the following:

First, one may divide processes into segments according to the boundaries of the critical sections and the boundaries of sections of code that have precedence relationships with sections of code of other processes. Thus, one may divide A into three segments A_0, A_1, and A_2, such that segment A_0 consists of the first 20 time units of computation of A, segment A_1 consists of the next 20 time units of computation of A, and segment A_2 consists of the last 20 time units of computation of A. There is no need to divide processes B, C, D, E, and F. In processes B, C, D, E, and F, the entire process can be considered as consisting of a single segment.[9]

In process A, there exists a critical section (A_0, A_1) which contains all use of the shared variable a and another critical section (A_1, A_2) which contains all use of the shared variable d. In process D, the critical section consisting of the entire process (D) and contains the use of d, a, and e. In process E, the critical section consisting of the entire process (E) and contains the use of e. In process F, the critical section consisting of the entire process (F) and contains the use of e, d, c, and a.

Thus,

$$(A_0,\ A_1)\ EXCLUDES\ (D), (D)\ EXCLUDES\ (A_0, A_1),$$
$$(A_0, A_1)\ EXCLUDES\ (F), (F)\ EXCLUDES\ (A_0, A_1),$$
$$(D)\ EXCLUDES\ (F), (F)\ EXCLUDES\ (D)$$

should be specified in order to prevent processes A, D, and F from simultaneously accessing a. Similarly,

$$(A_1, A_2)\ EXCLUDES\ (D), (D)\ EXCLUDES\ (A_1, A_2),$$
$$(A_1, A_2)\ EXCLUDES\ (F), (F)\ EXCLUDES\ (A_1, A_2),$$

in addition to $(D)\ EXCLUDES\ (F)$, $(F)\ EXCLUDES\ (D)$ specified above should be specified in order to prevent processes A, D, and F from simultaneously accessing d. Likewise, $(D)\ EXCLUDES\ (E)$, $(E)\ EXCLUDES\ (D)$, $(E)\ EXCLUDES\ (F)$, $(F)\ EXCLUDES\ (E)$, in addition to $(D)\ EXCLUDES\ (F)$, $(F)\ EXCLUDES\ (D)$ specified above, should be specified to prevent processes D, E, and F from simultaneously accessing e.

The critical section (A_1) in process A, the critical section (C) in process C, and the critical section (F) in process F contain all use of the shared variable c in the three processes. In order to prevent process A and process C from simultaneously accessing c, $(A_1)\ EXCLUDES\ (C)$ and $(C)\ EXCLUDES\ (A_1)$ should be specified. In order to prevent process C and process F from simultaneously accessing c, $(F)\ EXCLUDES\ (C)$, and $(C)\ EXCLUDES\ (F)$ should be specified. Note that $(A_0, A_1)\ EXCLUDES\ (F)$, $(F)\ EXCLUDES\ (A_0, A_1)$, specified earlier, will prevent process A and process F from simultaneously accessing c.

Note that, in process A, the critical section (A_0, A_1) overlaps with the critical section (A_1, A_2), while the critical section (A_1) is nested within both (A_0, A_1) and (A_1, A_2).

Because of the required producer/consumer relationship between process A and process C related to the use of shared variable b, $A_0\ PRECEDES\ C$ should be specified.

Because segments A_0, A_1, and A_2 all belong to the same process, $A_0\ PRECEDES\ A_1$ and $A_1\ PRECEDES\ A_2$ should be specified.

8. It is expected that PRECEDES relations will usually only be defined on segments of processes with identical periods.

9. Each segment i can be preempted at any time by segments that are not excluded by i.

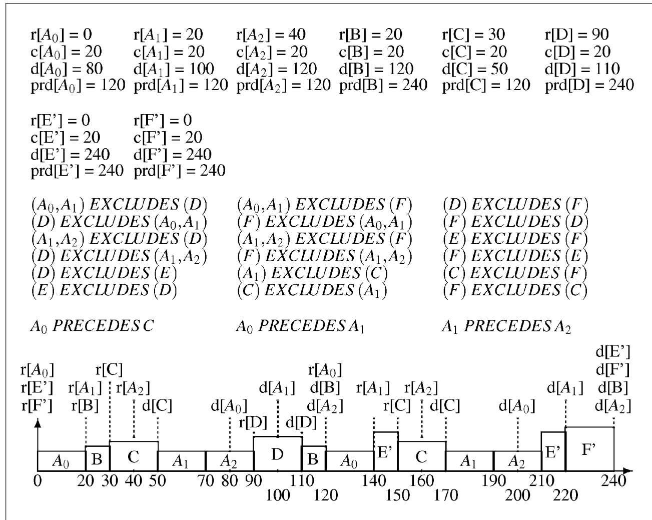


Fig. 2. A preruntime schedule for the four periodic processes A, B, C, and D, and the two asynchronous processes E and F.

The asynchronous processes E and F can be converted into new periodic processes as follows.

Using the set of conditions in the heuristic in Step 3 of the approach, asynchronous process E can be converted into a new periodic process E′ as follows:

- From 3a: $r[E'] = 0$.
- From 3b: $c[E'] = c[E] = 20$.
- From 3c: The existing set of periods of periodic process contains two period lengths, $prd[A] = prd[C] = 120$ and $prd[B] = prd[D] = 240$ time units. Among them, $prd[E'] = prd[B] = prd[D] = 240$ is the largest menber of the existing set of periods of periodic processes that satisfies the conditions in 3c, i.e., $2 \times prd[E'] - 1 = 2 \times 240 - 1 = 479 \leq d[E] = 480$ and $prd[E'] = 240 \leq min[E] = 242$.
- From 3d: $d[E'] = 240$ is equal to the largest integer such that $d[E'] + prd[E'] - 1 = d[E'] + 240 - 1 \leq d[E] = 480$ and $d[E'] = 240 \leq prd[E'] = 240$.

Finally, one can verify that the condition 3e holds: $d[E'] = 240 \geq c[E] = 20$.

Thus, $r[E'] = 0$, $c[E'] = 20$, $d[E'] = 240$, and $prd[A] = 240$.

Similarly, asynchronous process F can be converted into a new periodic process F′ such that:

$$r[F'] = 0, c[F'] = 20, d[F'] = 240, prd[F'] = 240.$$

Using the formulas in Step 4 of the approach, the release time and deadline of each segment in A can be calculated as follows:

$$r[A_0] = r[A] = 0, r[A_1] = r[A] + c[A_0] = 20,$$
$$r[A_2] = r[A] + c[A_0] + c[A_1] = 40;$$

$$d[A_0] = d[A] - (c[A_1] + c[A_2]) = 80,$$
$$d[A_1] = d[A] - c[A_2] = 100, d[A_2] = d[A] = 120.$$

There is now a total of two different process period lengths: $prd[A] = prd[C] = 120$ and $prd[B] = prd[D] = prd[E'] = prd[F'] = 240$. The length of the preruntime schedule is equal to the least common multiple (LCM) of all the periods, which is $\mathrm{LCM}(120, 240) = 240$.

Within the preruntime schedule length of 240 time units, there are two instances each of A_0, A_1, A_2, and C, and one instance each of B, D, E′, and F′ that need to be scheduled.

Given the excludes relation defined on overlapping critical sections and the precedes relation defined on process segments specified above, an algorithm that can schedule processes with overlapping critical sections, such as the algorithm described in [55], should be able to find the feasible schedule shown in Fig. 2 in which all instances of all the process segments of A_0, A_1, A_2, C, B, D, E′, and F′ occurring within the preruntime schedule length of 240 time units meet their deadlines, and all the specified exclusion relation and precedes relation are satisfied.

If the algorithm was unable to deal with processes with overlapping critical sections, then one would have to replace the nested critical section (A_1) with a larger critical section (A_0, A_1, A_2) which is obtained by combining all the critical sections that surround (A_1) and, instead of specifying $(C)\ EXCLUDES\ (A_1)$ and $(A_1)\ EXCLUDES\ (C)$, one must specify exclusions involving a larger critical section, i.e., $(A_0, A_1, A_2)\ EXCLUDES\ (C)$ and $(C)\ EXCLUDES\ (A_0, A_1, A_2)$. This would decrease the chances that a feasible schedule will be found.[10]

Furthermore, if the algorithm was unable to schedule processes with critical sections that consist of more than one separately schedulable segments, then, instead of specifying a precedes element $A_0\ PRECEDES\ C$ which involves a shorter single segment A_0, one must specify the precedes element $A\ PRECEDES\ C$, which involves a lengthier segment consisting of the whole process A. This would also decrease the chances that a feasible schedule will be found.

In Step 6, if the particular way of executing segments according to the preruntime schedule described in this paper is used, then, prior to runtime, the following should be done:

Context saving code should be appended at the end of the following segments and subsegments in the preruntime schedule (see Fig. 3): first instance of segment A_0 between time unit 0 and 19, second instance of segment A_0 between time unit 120 and 139, subsegment B between time unit 20 and 29, subsegment E′ between time unit 140 and 149.

Context restoring code should be inserted at the beginning of the following segments and subsegments: first instance of segment A_1 between time unit 50 and 69, second instance of segment A_1 between time unit 170 and 189, subsegment B between time unit 110 and 119, and subsegment E′ between time unit 210 and 219.

Code that transfers control to the dispatcher should be appended to the end of all the segments and subsegments in the preruntime schedule.

In the following, we will discuss how the preruntime scheduling approach makes it easier to inspect and verify the timing properties of software using the processes A, B,

10. In this particular case, one can easily verify that no feasible schedule such that all processes meet their deadlines exists when overlapping critical sections are disallowed.

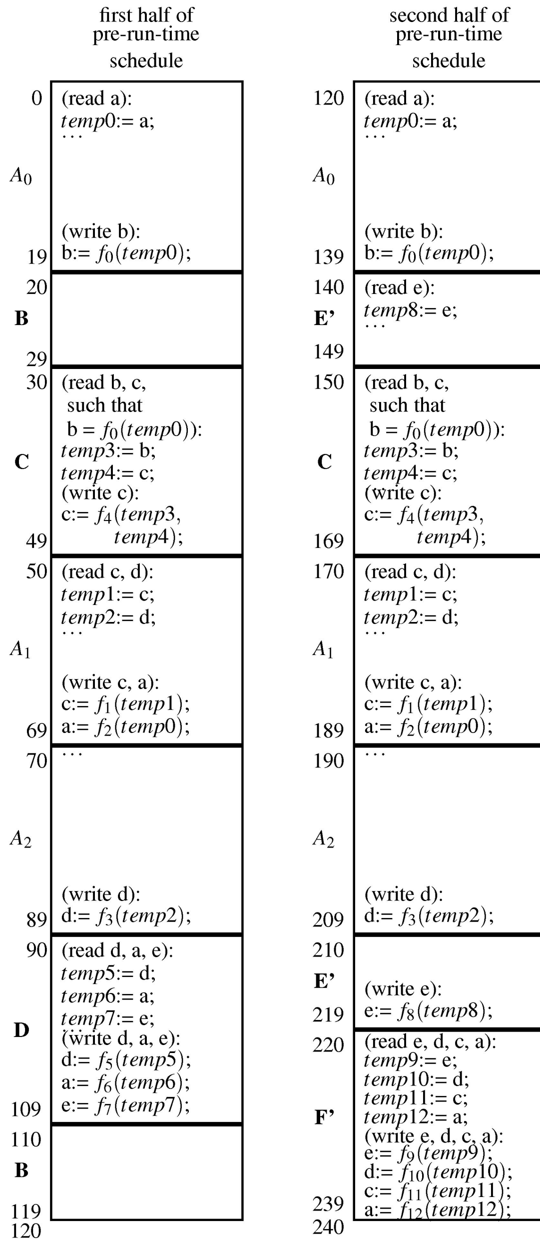


Fig. 3. The detailed code-layout in the preruntime schedule for the four periodic processes A, B, C, and D, and the two asynchronous processes E and F makes it much easier to inspect the timing and runtime behavior of the programs.

C, D, E, and F and the preruntime schedule in Fig. 3 as an example.

1. Instead of having to exhaustively analyze and inspect a huge number of different possible interleaving/concurrent task execution sequences, with a preruntime scheduling approach one only needs to inspect one single preruntime schedule each time.

   If we did not use preruntime scheduling, but had allowed processes A-F to execute nondeterministically at runtime, even if we assume that each of the processes A-F can only execute exactly one single instruction in each time unit, just the number of different possible interleavings of two instances each of processes A and C, and one instance each of B, D, E, and F within a time interval of 240 time units alone would be[11]

$$\frac{240!}{120! \times 20! \times 40! \times 20! \times 20! \times 20!}.$$

   One may argue that conventional non-real-time systems routinely include a much larger number of processes, and conventional non-real-time scheduling and synchronization methods seem to be able to handle them without too much of a problem, but a fundamental difference is that, in conventional non-real-time systems, most of the possible interleavings do not affect logical correctness, and, if some of them do affect logical correctness, in most cases we can err on the side of allowing too few interleavings, even prohibit interleavings altogether, without affecting logical correctness. More importantly, the exact times at which interleavings occur do not matter. In contrast, in a real-time system, in many cases, allowing or disallowing certain interleavings at precise times can be crucial for timing correctness. In Example 1, one may observe that, if either process C could not preempt one precise process (process B) at one precise time (time 29) or if any other process other than B were allowed to execute at any time between time 20 and time 29 (even though B does not contain any critical section), then it would be impossible to satisfy all the timing constraints.

   With the preruntime scheduling approach, to verify that all the timing constraints of processes A-F will always be satisfied, one only needs to inspect one single interleaving sequence of the processes —the single interleaving sequence shown in the preruntime schedule in Fig. 3. In comparison, *if processes A-F are to be scheduled at runtime, then to verify that all the timing constraints of processes A-F will always be satisfied, one would need to prove that every possible interleaving permitted by the runtime system of processes A-F does not violate any timing constraint.*

2. In each preruntime schedule, the interleaving/concurrent task execution sequence is statically and visually laid out in one straight line of code. This makes it much easier to verify, by straightforward visual inspection of the preruntime schedule, that all the timing constraints, such as release times and deadlines, periods, low jitter requirements, etc., are met by the execution sequence.

   Consider the reality that, at actual runtime, each of these processes could take any time between 0 and their worst-case computation time to complete even if each of them were run uninterrupted. When processes are scheduled at runtime, it is possible that some process finishing at some nondeterministic time, may cause a violation of a timing constraint. For example, if processes A-F are to be scheduled at

11. The number of interleavings given assumes that the instructions in each instance of each process are executed in one single order, the instructions in the two instances of processes A are executed in one single order, and the instructions in the two instances of processes C are executed in one single order.

runtime, then, if process B finishes early, say at time 25, then a runtime scheduler might schedule process F for execution at time 25 since process F is an asynchronous process that arrives at nondeterministic times and F could be the only process that is ready at that time. But, if process F starts executing at time 25, it could cause process C to miss its deadline because process F cannot be interrupted by process C.[12]

This shows that, in general, *if processes A-F are to be scheduled at runtime, then to verify that all the timing constraints of processes A-F will always be satisfied, one would also need to prove that for every possible computation time between 0 and the corresponding worst-case computation time under every possible interleaving permitted by the runtime system of processes A-F, no timing constraint will be violated.*

When a preruntime scheduling approach is used, we do not need to be concerned about the possible effects of variations in the actual execution time of the process on timing properties. Assuming that the worst-case computation times of process segments are estimated correctly, each process segment will always execute within the allocated time slot in the preruntime schedule, regardless of the actual execution time. By straightforward visual inspection of the preruntime schedule in Fig. 3, one can easily confirm that every process will always meet its deadline, regardless of how long any of the process segments actually execute at runtime.

In a recent attempt by researchers at NASA Ames Research Center and Honeywell to verify some time partitioning properties of the Honeywell DEOS real-time operating system scheduler kernel using the model checker SPIN [32], in order to limit the number of potential execution paths and avoid state space explosion, the researchers had to limit the choices as to the amount of time that a thread could execute to just three possibilities: Either it used no time, it used all of its time, or it used half of the time. Even with such limitations, they found the state space was too large to be exhaustively verified without using a high level of abstraction when they allowed dynamic thread creation and deletion, even though the kernel overhead, such as the time required for runtime scheduling and context switching, was not taken into account; the system only included a total of three threads—one main thread and two user threads [43], [15]!

3. Instead of using complex, unpredictable runtime synchronization mechanisms to prevent simultaneous access to shared data, the preruntime scheduling approach prevents simultaneous access to shared data simply by constructing preruntime schedules in which critical sections that exclude each other do not overlap in the schedule. This makes it easy to verify, by straightforward visual inspection of the preruntime schedule, that requirements such as exclusion relations and precedence relations between code segments of real-time tasks, are met by the execution sequence.

   For example, by straightforward visual inspection of the preruntime schedule in Fig. 3, one can easily confirm that process C will always correctly read the value of shared variable b produced by segment A_0, and no other process will be executing while process C accesses the shared variables b and c, and process C will always meet its deadline, regardless of how long any of the process segments actually execute at runtime.

   If processes A-F are to be scheduled at runtime, then, in order to satisfy their timing constraints, it would require that the runtime synchronization mechanism must simultaneously possess, at the very least, the following capabilities:

   a. the ability to handle process release times;
   b. the ability to handle precedence constraints between processes;
   c. the ability to handle overlapping or nested critical sections;[13]
   d. the ability to prevent deadlocks;[14]
   e. the ability to handle the type of situations shown in Step 3.b above, i.e., prohibit a process that is ready from executing, even if the CPU will be left idle for some interval of time, if there is a possibility that the immediate execution of that process may cause a second process that is not yet ready to miss its deadline in the future;
   f. the ability to maximize schedulability and processor utilization;
   g. at the same time provide a priori guarantee that the set of schedulable processes will always satisfy their timing constraints.

   We are not aware of any runtime synchronization mechanism that simultaneously possesses all of these capabilities, while the preruntime scheduling approach simultaneously possesses all of these capabilities.

   On the contrary, it can be observed that low-level implementation code of existing real-world synchronization and scheduling mechanisms that claim to

12. In fact, this example shows that none of the existing fixed priority scheduling algorithms, including the Priority Ceiling Protocol [45], would be able to guarantee that the timing constraints of C will never be violated, whereas we have already shown that the preruntime scheduling approach can easily provide such a guarantee.

13. In practice, it is quite important for a scheduling algorithm to be able to handle exclusion relations defined on overlapping critical sections; in many hard-real-time applications, many naturally occurring critical sections overlap. For example, the periodic processes in the US Navy's A-7E aircraft operational flight program [28], [24] contained many overlapping critical sections. If the algorithm used for scheduling processes does not allow overlapping critical sections, then one would be obliged to treat each set of such sections as a single critical section; this can significantly reduce the chances of finding a feasible schedule. One would not have been able to satisfy all the timing constraints and construct a feasible preruntime schedule for the US Navy's A-7E aircraft operational flight program if overlapping critical sections were not allowed.

14. In general, deadlock avoidance at runtime requires that the runtime synchronization mechanism be conservative, resulting in situations where a process is blocked by the runtime synchronization mechanism, even though it could have proceeded without causing deadlock. This in general reduces the level of processor utilization [57].

implement well-known high-level real-time synchronization protocols and algorithms, such as the Priority Ceiling Protocol [45], and Rate Monotonic Scheduling [36], often have very complex timing behaviors that are quite different from the theoretical properties that the high-level protocols and algorithms are supposed to have.

These runtime synchronization and scheduling mechanisms not only often incur high amounts of costly system overhead and do not have the capability to handle complex dependencies and constraints, they also often make it virtually impossible to accurately predict the system's timing behavior [56], [57].

These realities explain why up to the present time, virtually all large scale complex safety-critical real-time applications use some form of static schedule, even if the static schedules have to be manually constructed,[15] are very fragile and difficult to maintain, as in the case of cyclic executives [10], [38].

4. In a preruntime schedule, there is no possibility of task deadlocks occurring.

   With the preruntime scheduling approach, processes A-F are automatically executed at their scheduled start times; they do not encounter any synchronization mechanism that may block them.

   If processes A-F are to be scheduled at runtime, then they would require some kind of blocking synchronization mechanism to prevent them from accessing a shared variable in a critical section when another process is accessing the same shared variable in a critical section. If the runtime blocking synchronization mechanism does not prevent deadlocks, then it is possible that process A may start accessing a at the same time that process D accesses d. This will create a deadlock since process A would need to access d and process D would need to access a. Similarly, a second deadlock may happen involving process D and process F through accessing the shared variables d and e. A third deadlock may happen involving process A and process F through a and d. A fourth deadlock may happen involving process A and process F through c and d. A fifth deadlock may happen involving process A and process F through a and c.

   The contrast between preruntime scheduling and runtime scheduling of processes concerning the ability to avoid deadlocks is noteworthy:

   When processes are scheduled at runtime, in order to guarantee that no deadlocks will happen, one would have to check through all possible execution paths that may lead to deadlock, which, as one of the reviewers has noted, is often not tractable in practice on software.

   Interestingly, when the preruntime scheduling approach is used, in order to guarantee that no deadlocks will happen, it suffices for a preruntime scheduler to find just one single feasible preruntime schedule!

   This mirrors the contrast between preruntime scheduling and runtime scheduling on the ability to satisfy timing constraints mentioned in 1 and 2.

   With a preruntime scheduling approach, the proof of the preruntime schedule's timing correctness and deadlock free properties is obtained essentially through a "proof-by-construction" process—the search for one feasible schedule by the preruntime scheduler,[16] *which is fundamentally easier than a proof that must check through every possible executable path, as only one feasible schedule for each mode of system operation is needed*.

5. Instead of having to assume that context switches can happen at any time, it is easy to verify, by straightforward visual inspection of the preruntime schedule, exactly when, where, and how many context switches may happen.

   By inspecting the preruntime schedule one would also know exactly which information needs to be saved and later restored. This allows one to minimize the system overhead required with context switching.

   If processes A-F are to be scheduled at runtime, then it would be impossible to know exactly when, where and how many context switches may happen. Not only could there be a much higher number of context switches, but also, at every time there is a context switch at runtime, one would have no choice but to blindly save and restore every possible bit of information that *might* be affected. This can increase the system overhead substantially.

6. With a preruntime scheduling approach, one can switch processor execution from one process to another through very simple mechanisms, such as procedure calls, or simply by catenating code, which reduces system overhead and simplifies the timing analysis.

   In general, under the assumption that the worst-case execution time of each segment/subsegment has been estimated correctly,[17] when a single

15. It is also noted here that previous work on static scheduling [49], [50] does not possess the capability of handling all the constraints and dependencies mentioned in Steps 1-4 above, thus they are not really suitable for automating the task of constructing preruntime schedules. Most of the work on static scheduling either assumes that all processes are completely preemptable, which makes it impossible to prevent simultaneous access to shared resources; assumes that none of the tasks are preemptable, which significantly reduces the chances of finding a feasible preruntime schedule; or they are incapable of handling release times, precedence relations, or overlapping critical sections, which are required in many real-time applications.

16. When critical sections overlap, the preruntime scheduling algorithm needs to detect and handle deadlocks while searching for a feasible schedule. However, this is easily accomplished by using a standard resource allocation graph technique and checking for cycles involving processes and critical sections in the graph. If a cycle occurs when a resource is "allocated," i.e., when the scheduler constructs part of a schedule in which it simulates allocating processor or other resources, the scheduler can simply discard the partially built schedule as unfeasible and start trying to construct an alternative feasible schedule which is different from the discarded partial schedule. Since all of this is simulation that is performed offline, the time needed to detect and handle deadlocks by the preruntime scheduler is not critical, unlike the case of a scheduler allocating real resources during runtime.

17. One should note that, while catenating a sequence of segments/subsegments reduces system overhead, it may also make it difficult for the dispatcher to immediately detect cases where a segment/subsegment executes longer than its allocated time slot in the preruntime schedule, possibly resulting in a missed deadline not being immediately detected by the dispatcher.

processor is used,[18] a continuous sequence of segments/subsegments can be catenated when all of the release times of the segments/subsegments in the sequence are less than or equal to the release time of the first segment/subsegment in the sequence.

A straightforward inspection of the preruntime schedule in Fig. 3 would allow one to conclude that it is possible to directly catenate the following sequences of segments (each sequence is enclosed in parentheses below) prior to runtime, while guaranteeing that no timing constraints or errors will occur:

Catenate (first instances of C, A_1, and A_2); Catenate (D, second subsegment of B); Catenate (second instance of A_0, first subsegment of E′); Catenate (second instances of C, A_1 and A_2, second subsegment of E′, F′).

(Note that the code for restoring context mentioned above will still need to be included in the catenations).

This will reduce system overhead as the dispatcher only needs to be invoked at the start and end times of sequences of segments/subsegments instead of at the start and end time of every individual segment/subsegment.

7. With a preruntime scheduling approach, an automated preruntime scheduler can help automate and speed up important parts of the system design and inspection process. Whenever a program needs to be modified, a new preruntime schedule can be automatically and quickly generated, allowing one to quickly learn whether any timing requirements are affected by the modifications.

   As preruntime schedules can be carefully designed before runtime, the designer has the flexibility to take into account many different possible scenarios of system errors and tailor different strategies in alternative schedules to deal with each of them. For example, in the case of system overload, an alternative preruntime schedule which only includes the set of processes that are considered to be essential under the particular circumstances can be executed.

   Performing modifications to the system online is also not difficult. One can easily insert code in preruntime schedules that, when activated by an external signal, will cause processor execution to switch from a previously designed preruntime schedule to a newly designed preruntime schedule that includes new processes and new functionality to meet new requirements during runtime.
8. Preruntime scheduling can make it easier to implement a relatively constant "loop time" for control systems software, making it easier to implement and verify timing properties that span longer durations by making use of the loop time; it can also help to reduce jitter in the output and guarantee constant sampling times for inputs both of which can be critical to guarantee stability of feedback control systems.
9. With a preruntime scheduling approach, most asynchronous processes can be converted to periodic processes so that, instead of using interrupts to respond to external and internal events, periodic processes are used to handle those events. This removes a significant source of nondeterministic behavior from the system, which greatly reduces the complexity of the software and its timing behaviors. Many researchers, experienced engineers, and practitioners have long considered interrupts to be the most dangerous [35], most error-prone, and most difficult to test and debug part of real-time software [25], [5], [6].

We have examined a number of hard real-time applications and it appears that, in most of them, periodic processes constitute the bulk of the computation. Usually, periodic processes have hard deadlines, whereas asynchronous processes are relatively few in number and usually have very short computation times. We believe that this is because most hard real-time applications are concerned with simulating continuous behavior, and simulating continuous behavior requires many computationally intensive periodic actions. The sporadic events are relatively rare compared with the periods of the processes. In addition, information about most external and internal events can be buffered until it can be handled by periodic processes. A preruntime scheduling approach should be applicable to most such hard real-time applications in which it is absolutely necessary to provide an a priori guarantee that all hard timing constraints will be satisfied.

For example, in order to investigate the feasibility of applying a preruntime scheduling approach to an existing hard real-time system, the F/A-18 aircraft (referred to as the CF-188 by the Canadian Forces) Mission Computer (MC) Operational Flight Program (OFP), a detailed analysis and documentation of the computational requirements and timing constraints of the CF-188 aircraft MC OFP, was conducted by Shepard and Gagne, in part at the CF-188 Weapon Software Support Unit at Canadian Forces Base Cold Lake, Alberta. The following description of the results of the CF-188 MC OFP project is based on [46], [47]. In the CF-188 MC OFP, most processes were periodic and were grouped into four separate task rates: 20 Hz, 10 Hz, 5 Hz, and 1 Hz. It was found that most routines referred to as "demand" (asynchronous) routines in the original documentation were invoked by periodic monitor interrupts, which meant that they were actually periodic; while the remaining demand routines could also be transformed into periodic processes and effectively handled by polling because the computation load associated with the demand routines was very small. Thus, it was found that approximately 95 percent of the existing interrupts could be eliminated. Aside from the interrupts required to synchronize the real-time clocks, the only interrupts which were maintained were the ones associated with software or hardware faults. The retention of

18. In cases where more than one processor is used, if some segment s has an exclusion relation or precedence relation with some other segment on a different processor that may be violated if s starts earlier than the beginning of its time slot, then segment s should not be catenated with segments that precede it.

these interrupts did not create a runtime implementation problem since their occurrence must terminate the normal execution of the schedules in order to initiate fault recovery actions. It was concluded that implementing a preruntime scheduling strategy for a hard real-time system such as the CF-188 MC system was a suitable and viable option and that, in addition to many important benefits such as reducing context switching overhead, reducing the cost and difficulties associated with software maintenance, and facilitating effective life cycle management of the CF-188 MC system, including the capability to assess the software growth capacity of the system,[19] "the single most important benefit provided by preruntime scheduling technology is a way to verify and guarantee that all the timing constraints will be observed at runtime." It was noted that the existing software for the CF-188 MC OFP, which used a rate monotonic, preemptive scheduling discipline based on clock interrupts, did not provide such a guarantee, even when extensive testing was carried out, and that specific occurrences of timing constraint violations in the existing software following "verification via simulations" had been reported within the United States Navy F/A-18 community.

In order to apply preruntime scheduling, periodic processes were extracted from the existing cycles of the task rates, and the existing input-processing-output order within each task rate was conserved by the specification of precedence constraints between the processes. Special programs were created and used to extract precedence and exclusion constraints from the CF-188 MC OFP source code.[20] Two variable lists were generated for each of 2,324 routines in the source code. One list contained all the variables written to by the routine. The other list included all the variables read by the routine. These lists were then correlated in order to determine the precedence and exclusion constraints between all possible pairs of routines. Based on the existing routine sequence, if a routine was found to write to a variable which was read by a routine succeeding it in the same sequence on every execution of the task rate, a precedence constraint was specified. An exclusion relation was specified in cases where a precedence constraint was not needed but preventing read-write problems was necessary. This resulted in 620 synchronization constraints, including 555 precedence constraints and 65 exclusion constraints[21] on 675 process instances in a preruntime schedule length of 1,000 milliseconds. The precedence constraints were used to divide the routines in the source code into processes. Where the precedence constraints linked a group of routines as a single execution thread (no parallel paths possible), these routines were merged into a single process. When a routine preceded two or more processes, a separate process was defined in order to reflect the possibility of parallelism. The preruntime scheduler was able to generate a feasible schedule in just one iteration in 58 seconds. While resource limitations and operational considerations precluded the actual reimplementation of the OFP, this exercise nevertheless demonstrated the feasibility and the many important potential benefits of applying a preruntime scheduling approach to the F/A-18 aircraft Mission Computer software [46], [47].

Using a preruntime scheduling approach has also been found to be feasible and to have many similar important advantages for the US Navy's A-7E aircraft operational flight program, even when the preruntime scheduler was not yet automated and preruntime scheduling had to be performed by hand [28], [24].

Readers are directed to [57], [52] for discussion on other advantages and perceived disadvantages of the pre-runtime scheduling approach not discussed in this paper due to space limitations.

5 How the Preruntime Scheduling Approach Alleviates the Theoretical Limitations and Practical Difficulties

In the following, we will discuss how the preruntime scheduling approach helps to alleviate the theoretical limitations and practical difficulties in verifying timing properties of software that were mentioned in Section 2.

5.1 Alleviating the Theoretical Limitations

1. Preruntime scheduling reduces the complexity of the timing behaviors of the software:
   a. By restricting the number of different possible relative orderings of the runtime process segment executions in each Least Common Multiple of the periods of the set of periodic processes (including asynchronous processes that are converted to periodic processes) in each system mode to be a relatively small constant in relation to the number of process segments, the number of global timed states that needs to be checked during inspection and verification is significantly reduced.
   b. By avoiding the need for complex runtime synchronization mechanisms and process blocking through the use of exclusion relations and precedence relations on ordered pairs of process segments, both the number and complexity of various properties that need to be proven, including various safety and liveness properties, e.g., absence of simultaneous access to shared resources, absence of deadlocks, (time bounded)

19. It was found that the a priori knowledge of the location and length of the gaps in the preruntime schedule can facilitate the specifications of the timing requirements for new processes or help assess the feasibility of increasing the computation time of existing processes.

20. "Reverse engineering" to extract precedence and exclusion constraints from the existing CF-188 MC OFP source code was necessary because of unavailability of necessary documentation. Because of the unsystematic shuffling of routines to finetune the OFP, the existing routine sequence within each processing cycle could not be assumed to reflect the correct precedence order between the routines. Instead, the dependencies between the routines had to be determined.

21. It was reported that some precedence constraints were specified that should have been exclusion constraints. This resulted in simplification of the scheduling problem, even though this creates the possibility that some solutions which might be feasible were not investigated. In this case, it did not create a problem because a feasible schedule was easily found. If a precedence relation is used when an exclusion relation is sufficient, it will still prevent simultaneous access to shared resources. However, the preruntime scheduler will have less flexibility in finding a feasible schedule.

guaranteed access to resources, (time bounded) termination of parts of the software, etc., are significantly reduced.

2. Preruntime scheduling makes it possible to compartmentalize each process segment's possible timing/logical errors within specific time frames in the preruntime schedule. The ability to check timing constraints at predetermined points in the preruntime schedule helps prevent timing unpredictability due to both runtime logical errors and timing errors. The preruntime schedule provides precise knowledge about which other process segments could be affected by any process' possible timing/logical errors and the times at which they could be affected. With such knowledge, more effective fault-tolerant procedures, including alternative preruntime schedules, can be incorporated. This should simplify the task of verifying critical timing properties of the software.
3. The concept of cooperating sequential processes provides a powerful and well-understood abstraction for structuring concurrent software. With the number of different timing behaviors significantly reduced through preruntime scheduling, there should be much less of a need to use other less well-understood forms of approximation or abstraction.

5.2 Alleviating Practical Difficulties

1. With preruntime scheduling, asynchronous processes can be converted into periodic processes, eliminating a source of nondeterminism, which simplifies verification of timing properties by significantly reducing the number of timed states that need to be checked.
2. With preruntime scheduling, exact knowledge about the runtime execution sequence will be known in advance, making it feasible to predict the worst-case execution time more accurately.
3. With preruntime scheduling, only a small constant number of different execution orderings needs to be studied, thus studying each of those orderings at the most detailed machine code/assember should be less constrained by time and space limits.

Thus, the preruntime scheduling approach reduces the complexity of inspection and verification of timing properties in all the cases of theoretical limitations and practical difficulties discussed in Section 2.

6 Conclusion

In most cases, researchers verify the timing properties of specifications/models/algorithms/protocols, but not actual code. In most cases, no proof is given that the specifications/models/algorithms/protocols have the same timing properties as the actual code.

A major benefit of the preruntime scheduling approach is that it allows one to systematically inspect and verify the timing properties of *actual code*, not just various high-level abstractions of code.

A preruntime scheduling approach combines two essential factors that we believe are key to allowing one to systematically structure large scale software with hard timing requirements so that it would be easier to inspect and verify the timing properties of the actual code of the software:

1. With the preruntime scheduling approach, the proof of the preruntime schedule's timing correctness and deadlock free properties is obtained essentially through a "proof-by-construction" process—the search for one feasible schedule by the preruntime scheduler, which is fundamentally easier than a proof that must check through every possible executable path.

   Once a preruntime schedule has been constructed, the task execution sequence is statically and visually laid out in one straight line of actual code. This makes it much easier to verify, through straightforward visual inspection of the actual code in the preruntime schedule, that all timing constraints, no matter how complex, are satisfied.

   Because of this fundamental reduction in the complexity of verifying the timing correctness and deadlock free properties, we believe that, compared with scheduling strategies and mechanisms that perform synchronization and scheduling at runtime, the preruntime scheduling approach has far better capability to scale-up to large systems.
2. The preruntime scheduling approach has the capability to simultaneously and systematically handle complex dependencies and constraints such as release times, exclusion relations defined on overlapping critical sections, precedence relations, that are often required by large scale, complex, safety-critical real-time applications, etc., that neither scheduling strategies and mechanisms that perform scheduling at runtime nor previous work on static scheduling can simultaneously and systematically handle. The capability of systematically handling complex dependencies and constraints is fundamental to allowing one to *automate* the tedious, error prone manual process of constructing preruntime schedules containing the actual code that, once constructed, would be much easier to inspect and verify.

Overly complex interactions between system components that execute in parallel is a main factor in creating an exponential blowup in complexity, especially in large scale, complex, nonterminating concurrent real-time software. The preruntime scheduling approach effectively reduces complexity by structuring real-time software as a set of cooperating sequential processes and imposing strong restrictions on the interactions between the processes.

Earlier, we mentioned that one of the reasons that program timing correctness in general is undecidable is because program logical correctness is in general undecidable, and timing correctness depends on logical correctness. Thus, even though each system component in the preruntime scheduling approach is much simpler because it is

restricted to be a sequential program where concurrency has been eliminated, we still need to ensure that each sequential program's worst-case execution time and logical correctness is decidable and can be independently verified. Although it may be infeasible to verify the logical and timing correctness of any *arbitrary* piece of software, one should be able to verify the logical and timing correctness of each sequential program if each sequential program has been methodically designed and structured to make its logical and timing properties easier to inspect and verify. We believe that a key to ensuring that the worst-case execution time and logical correctness of each sequential program is decidable and can be independently verified is to also impose strong restrictions on the sequential program structures. If the system is safety-critical and time-critical and if certain structures or methods of constructing any system component, including both hardware and software components, may make the logical and timing correctness of the system either undecidable or practically impossible to inspect and verify, then system designers should refrain from using those structures, methods, and components, and only use structures, methods, and components that result in systems that are predictable, verifiable, and easy to inspect. We plan to discuss these issues in detail in a future paper.

Acknowledgments

The author would like to thank the reviewers for numerous thoughtful comments and helpful suggestions on how to improve this paper. The author is indebted to Dave Parnas for helpful discussions and advice, related to this work. This work was partially supported by a Natural Sciences and Engineering Council of Canada grant. A preliminary version of this paper was presented at the First Workshop on Inspection in Software Engineering (WISE'01), Paris, France, July 21, 2001.

References

[1] R. Alur, C. Courcoubetis, T.A. Henzinger, and P.-H. Ho, "Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems," *Hybrid Systems,* R.L. Grossman, A. Nerode, A. Ravn, and H. Rischel, eds., 1993.

[2] R. Alur and D.L. Dill, "A Theory of Timed Automata," *Theoretical Computer Science,* vol. 126, pp. 183-235, 1994.

[3] R. Alur and T.A. Henzinger, "Real-Time Logics: Complexity and Expressiveness," *Proc. Fifth Ann. IEEE Symp. Logic in Computer Science,* pp. 390-401, 1990.

[4] R. Alur and T.A. Henzinger, "Logics and Models of Real Time: A Survey," *Real Time: Theory in Practice,* J.W. de Bakker et al., eds., 1992.

[5] S.R. Ball, *Debugging Embedded Microprocessor Systems.* Newnes, 1998.

[6] S.R. Ball, *Embedded Microprocessor Systems: Real World Design,* second ed. Newnes, 2000.

[7] P. Bellini, R. Mattolini, and P. Nesi, "Temporal Logics for Real-Time System Specification," *ACM Computing Surveys,* vol. 32, no. 1, pp. 12-42, 2000.

[8] A. Burns, K. Tindell, and A. Wellings, "Effective Analysis for Engineering Real-Time Fixed Priority Schedulers," *IEEE Trans. Software Eng.,* vol. 21, pp. 475-480, 1995.

[9] S.V. Campos and E.M. Clarke, "The Verus Language: Representing Time Efficiently with BDDs," *Theoretical Computer Science,* 1999.

[10] G.D. Carlow, "Architecture of the Space Shuttle Primary Avionics Software System," *Comm. ACM,* vol. 27, pp. 926-936, Sept. 1984.

[11] E.M. Clarke, E.A. Emerson, and A.P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Trans. Programming Languages and Systems,* vol. 8, pp. 244-263, 1986.

[12] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Progress on the State Explosion Problem in Model Checking," *Informatics. 10 Years Back. 10 Years Ahead,* R. Wilhelm, ed., 2001.

[13] E.M. Clarke and J.M. Wing, "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys,* Dec. 1996.

[14] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venter, D. Weil, and S. Yovine, "TAXYS: A Tool for the Development and Verification of Real-Time Embedded Systems," *Proc. 13th Int'l Conf. Computer Aided Verification (CAV),* pp. 391-395, 2001.

[15] D. Cofer and M. Rangarajan, "Formal Modeling and Analysis of Advanced Scheduling Features in an Avionics RTOS," *Proc. Int'l Workshop Embedded Software (EMSOFT 2002),* A. Sangiovanni-Vincentelli and J. Sifikis, eds., 2002.

[16] P. Cousot, "Abstract Interpretation Based Formal Methods and Future Challenges," *Informatics. 10 Years Back. 10 Years Ahead,* R. Wilhelm, ed., 2001.

[17] P. Cousot and R. Cousot, "Verification of Embedded Software: Problems and Perspectives," *Proc. Int'l Workshop Embedded Software (EMSOFT 2001),* 2001.

[18] C. Daws, A. Olivero, S. Tripakis, and S. Yovine, "The Tool KRONOS," *Hybrid Systems III, Verification and Control,* 1996.

[19] E.W. Dijkstra, "Cooperating Sequential Processes," *Programming Languages,* F. Genuys, ed., pp. 43-112, 1968.

[20] D. Dill and J. Rushby, "Acceptance of Formal Methods: Lessons from Hardware Design," *Computer,* vol. 29, pp. 23-24, 1996.

[21] B. Dutertre, "Formal Analysis of the Priority Ceiling Protocol," *Proc. IEEE Real-Time Systems Symp.,* pp. 151-160, Nov. 2000.

[22] E.A. Emerson and E.M. Clarke, "Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons," *Science of Computer Programming,* vol. 2, pp. 241-266, 1982.

[23] E.A. Emerson, A.K. Mok, A.P. Sistla, and J. Srinivasan, "Quantitative Temporal Reasoning," *Proc. Second Workshop Computer-Aided Verification,* pp. 136-145, 1989.

[24] S.R. Faulk and D.L. Parnas, "On Synchronization in Hard-Real-Time Systems," *Comm. ACM,* vol. 31, pp. 274-287, Mar. 1988.

[25] J.G. Ganssle, *The Art of Designing Embedded Systems.* Newnes, 2000.

[26] G. Ghezzi, D. Mandrioli, S. Morasea, and M. Pezze, "A Unified High-Level Petri Net Formalism for Time-Critical Systems," *IEEE Trans. Software Eng.,* vol. 17, no. 2, pp. 160-172, Feb. 1991.

[27] C. Heitmeyer, "On the Need for Practical Formal Methods," *Proc. Fifth Int'l Symp. Formal Techniques in Real-Time and Fault-Tolerant Systems,* 1998.

[28] K. Heninger, J. Kallander, D.L. Parnas, and J. Shore, "Software Requirements for the A-7E Aircraft," NRL Report No. 3876, Nov. 1978.

[29] T. Henzinger, Z. Manna, and A. Pneuli, "Temporal Proof Methodologies for Real-Time Systems," *Proc. 18th ACM Symp. Principles of Programming Languages,* pp. 353-366, 1991.

[30] T.A. Henzinger and J.-F. Raskin, "Robust Undecidability of Timed and Hybrid Systems," *Proc. Third Int'l Workshop Hybrid Systems: Computation and Control,* 2000.

[31] *Software Fundamentals: Collected Papers by David L. Parnas.* D.M. Hoffman and D.M. Weiss, eds., Addison-Wesley, 2001.

[32] G. Holzmann, "The Model Checker SPIN," *IEEE Trans. Software Eng.,* vol. 23, pp. 279-295, 1997.

[33] F. Jahanian and A. Mok, "Safety Analysis of Timing Properties in Real-Time Systems," *IEEE Trans. Software Eng.,* vol. 12, pp. 890-904, 1986.

[34] M.H. Klein, T. Ralya, B. Pollak, R. Obenza, and M.G. Harbour, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems.* Kluwer Academic, 1993.

[35] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer Academic, 1997.

[36] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. ACM,* vol. 20, Jan. 1973.

[37] K.L. McMillan, *Symbolic Model Checking.* Kluwer Academic, 1993.

[38] M.P. Melliar-Smith and R. L. Schwartz, "Formal Specification and Mechanical Verification of SIFT: A Fault-Tolerant Flight Control System," *IEEE Trans. Computers,* vol. 31, no. 7, pp. 616-629, July 1982.

[39] A.K. Mok, "Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment," PhD thesis, Dept. of Electrical Eng. and Computer Science, Massachusetts Inst. of Technology, Cambridge, Mass., May 1983.
[40] A.K. Mok, "The Design of Real-Time Programming Systems Based on Process Models," *Proc. IEEE Real-Time Systems Symp.*, pp. 5-17, Dec. 1984.
[41] J. Ostroff, *Temporal Logic of Real-Time Systems.* Research Studies Press, 1990.
[42] D.L. Parnas, "Inspection of Safety-Critical Software Using Program-Function Tables," *Proc. IFIP World Congress 1994,* vol. 3, Aug. 1994.
[43] J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger, "Verification of Time Partitioning in the DEOS Scheduler Kernel," *Proc. Int'l Conf. Software Eng.,* pp. 488-497, 2000.
[44] J. Queille and J. Sifakis, "Specification and Verification of Concurrent Systems in CAESAR," *Proc. Fifth ISP Conf.,* 1982.
[45] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. Computers,* vol. 39, no. 9, Sept. 1990.
[46] T. Shepard and M. Gagne, "A Model of the F-18 Mission Computer Software for Preruntime Scheduling," *Proc. 10th Int'l Conf. Distributed Computing Systems,* pp. 62-69, 1990.
[47] T. Shepard and M. Gagne, "A Preruntime Scheduling Algorithm for Hard-Real-Time Systems," *IEEE Trans. Software Eng.,* vol. 17, no. 7, pp. 669-677, July 1991.
[48] W. Soukoreff, "A Comparison of Two Preruntime Schedule-Dispatchers for RTAI," COSC6390 Project Report, Dept. of Computer Science, York Univ., Dec. 2001.
[49] "Hard Real-Time Systems," *IEEE Computer Society Tutorial,* J. Stankovic and K. Ramamrithan, eds., 1988.
[50] J. Stankovic, M. Spuri, M. DiNatale, and G. Buttazzo, "Implications of Classical Scheduling Results for Real-Time Systems," *Computer,* vol. 28, no. 6, pp. 16-25, June 1995.
[51] J. Xu, "Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence, and Exclusion Relations," *IEEE Trans. Software Eng.,* vol. 19, no. 2, Feb. 1993.
[52] J. Xu, "Making Timing Properties of Software Easier to Inspect and Verify," *IEEE Software,* July/Aug. 2003.
[53] J. Xu and K.-y. Lam, "Integrating Runtime Scheduling and Preruntime Scheduling of Real-Time Processes," *Proc. 23rd IFAC/IFIP Workshop Real-Time Programming,* June 1998.
[54] J. Xu and D.L. Parnas, "Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations," *IEEE Trans. Software Eng.,* vol. 16, no. 3, pp. 360-369, Mar. 1990.
[55] J. Xu and D.L. Parnas, "Preruntime Scheduling of Processes with Exclusion Relations on Nested or Overlapping Critical Sections," *Proc. 11th Ann. IEEE Int'l Phoenix Conf. Computers and Comm. (IPCCC-92),* Apr. 1992.
[56] J. Xu and D.L. Parnas, "On Satisfying Timing Constraints in Hard-Real-Time Systems," *IEEE Trans. Software Eng.,* vol. 19, no. 1, pp. 1-17, Jan. 1993.
[57] J. Xu and D.L. Parnas, "Fixed Priority Scheduling versus Pre-Run-Time Scheduling," *Real-Time Systems,* vol. 18, pp. 7-23, Jan. 2000.

Jia Xu received the Docteur en Sciences Appliquées degree in computer science from the Université Catholique de Louvain, Belgium, in 1984. From 1984 to 1985, he was a postdoctoral fellow at the University of Victoria, British Columbia, Canada. From 1985 to 1986, he was a postdoctoral fellow at the University of Toronto, Toronto, Ontario, Canada. He is currently an associate professor of computer science at York University, Toronto, Ontario, Canada. His current research interest is in real-time and embedded system engineering.

▷ **For more information on this or any computing topic, please visit our Digital Library at** http://computer.org/publications/dlib.